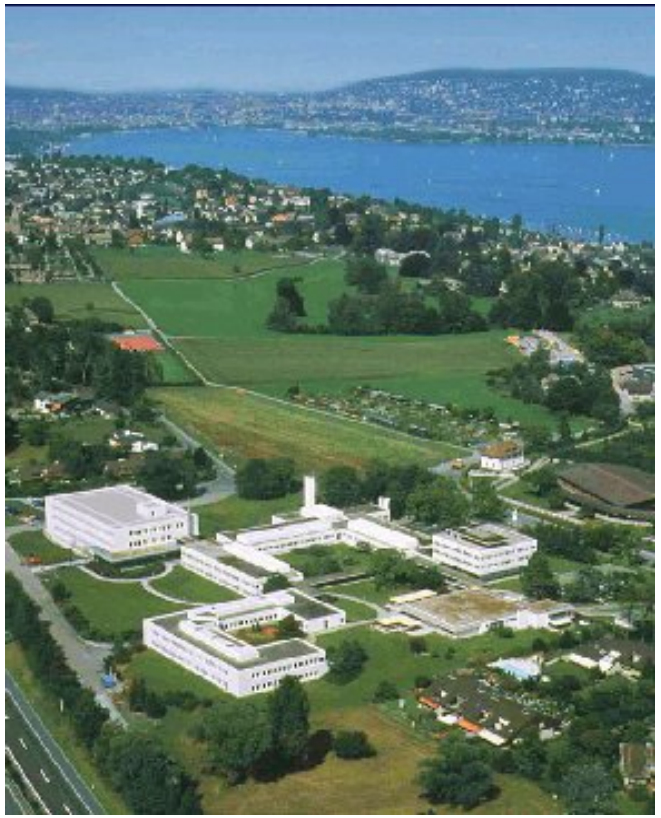
Introduction to Parallel Computing: The Message Passing, Shared Memory and Hybrid paradigms



C. Bekas
IBM, Zurich Research Laboratory
Rueschlikon 8803, Switzerland

Outline

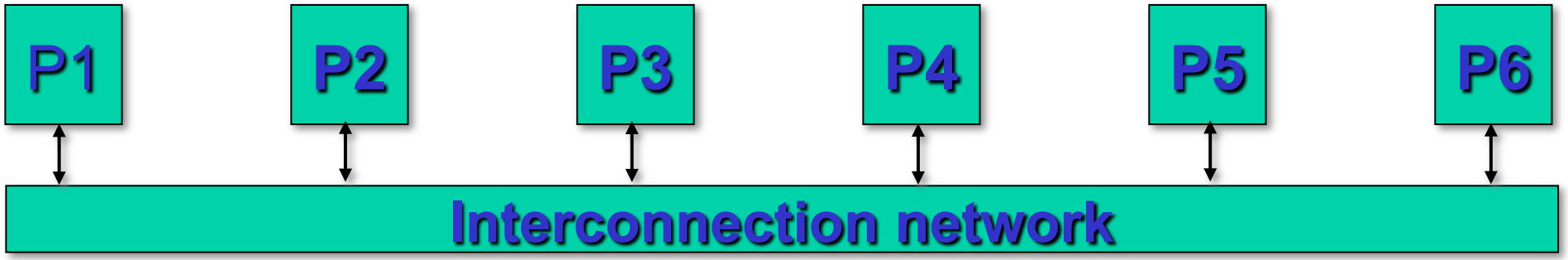
- ✓ Introduction to parallel architectures
- ✓ Quick reminder of parallel programming paradigms
  - Message passing (MPI)
  - Shared memory (OpenMP)
- ✓ Hybrid architectures
  - SMP clusters
  - Examples
- ✓ Hybrid programming models
  - Master thread only communications
  - Load balance issues and examples
  - Overlapping computation/communication

Parallel Programming: Software support & models

- Predominant Operating Systems
  - Linux, AIX, Solaris, HP-UX and Windows
- Programming Languages
  - Fortran 95, 90, 77
  - C, C++
  - New parallel languages: UPC, Chapel, X10
- Tools
  - Compilers
  - Debuggers
  - Performance Analysis Tools
- Parallel Models
  - **OpenMP - fine grain parallelism**
  - **MPI – coarse grain parallelism**
  - **Hybrid Models**
- Scientific Libraries
- Job schedulers
- Parallel File Systems

Parallel Programming Models: Distributed Memory

Machine architecture dictates the programming model



- ✓ Each processing element P has its own local memory hierarchy
- ✓ Local memory is not directly remotely accessible by other processing elements
  - ✓ Variants: One sided communication by remote direct memory access (RDMA)
- ✓ Processing elements are connected by means of a special network

Architecture dictates:

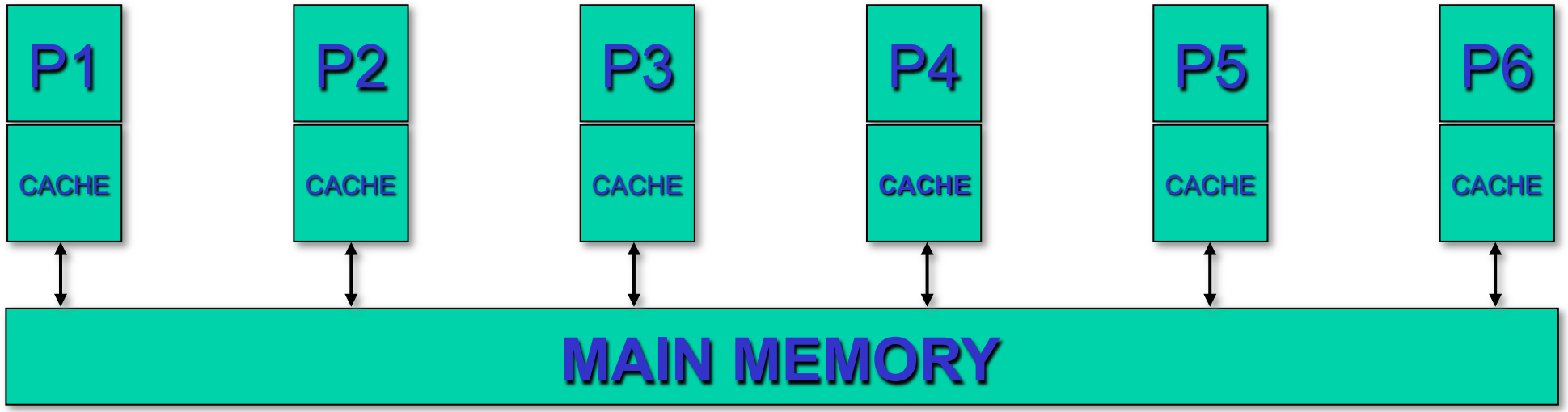
- ✓ Data and computational load must be explicitly distributed by the programmer
- ✓ Communication (data exchange) is achieved by messages
- ✓ Probably the oldest paradigm. Several variants: **PVM (Parallel Virtual Machine)**, **MPI (Message Passing Interface - ultimate winner)**
- ✓ *Loosely coupled variant for Big Data: MapReduce/Hadoop*

MPI Example: adding two vectors

```
#include “mpi.h” /* Include MPI header file */
int main(int argc, char **argv){
   int rnk, sz, n, i;
   double *x, *y, *buff;
   n = atoi(argv[1]); /* Get input size */

   MPI_Init(&argc, &argv); /* Initialize parallel environment */
   MPI_Comm_rank(MPI_COMM_WORLD, &rnk); /* Get my rank (MPI ID) */
   MPI_Comm_size(MPI_COMM_WORLD, &sz); /* Find out how many procs */
   chunk = n / sz; /* Assume sz divides n exactly */
   .
   .
   .
   MPI_Scatter(buff,chunk,MPI_DOUBLE,x,chunk,MPI_DOUBLE,0,MPI_COMM_WORLD);
   MPI_Scatter(&buff[n],chunk,MPI_DOUBLE,y,chunk,MPI_DOUBLE,0,MPI_COMM_WORLD);

   for (i=0; i<chunk; i++) x[i] = x[i] + y[i];
   MPI_Gather(x,chunk,MPI_DOUBLE,buff,chunk,MPI_DOUBLE,0,MPI_COMM_WORLD);

   MPI_Finalize();
}
```

Parallel Programming Models: Shared Memory

Machine architecture dictates the programming model



- ✓ **Processing elements share memory (either directly or indirectly)**
- ✓ **Communication among processing elements can be achieved by *carefully reading and writing in main memory***
- ✓ **Data and load distribution can be hidden from the programmer**
- ✓ **Messages can be implemented in memory as well (MPI)**
- ✓ ***Programming Model. OpenMP: Directives and Assertions***

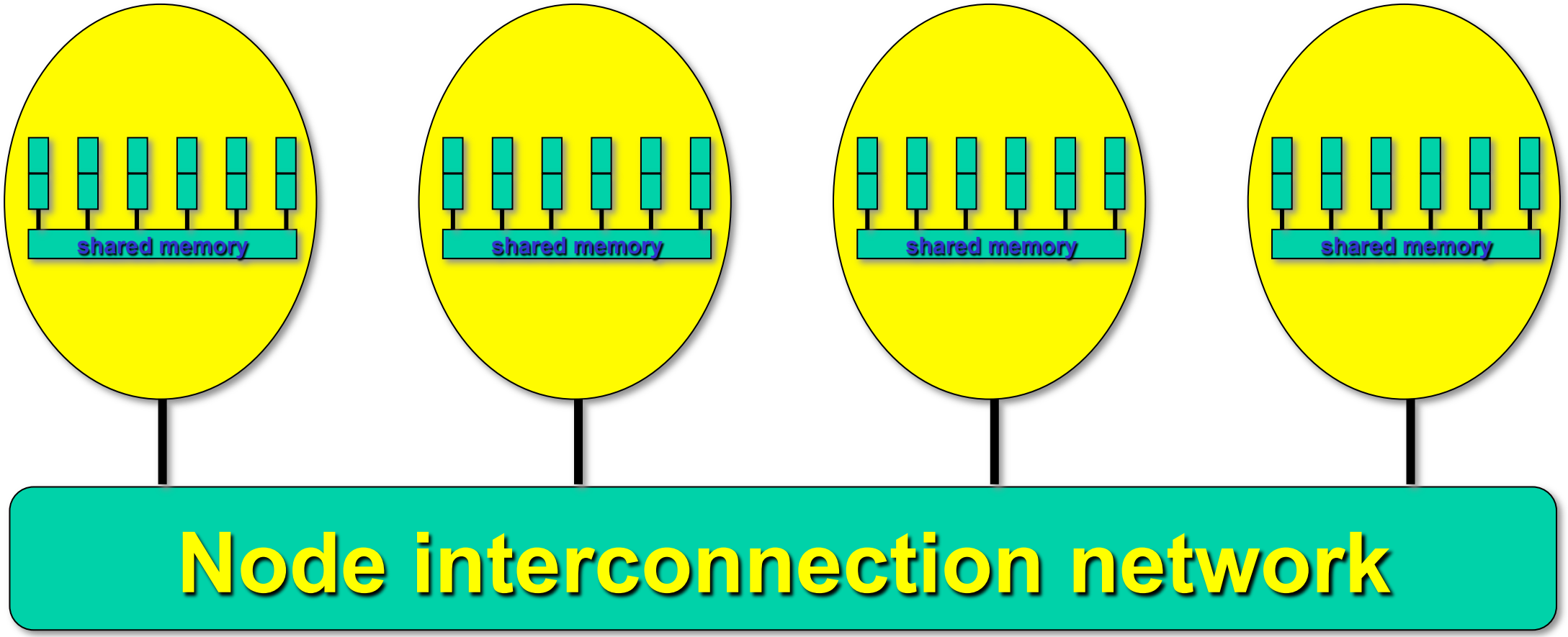
OpenMP Example: adding two vectors

```
int main(int argv, char **argv){

   int n, i;
   double *x, *y, *buff;
   n = atoi(argv[1]); /* Get input size */

   x = (double *)malloc(n*sizeof(double));
   y = (double *)malloc(n*sizeof(double));

   #pragma omp parallel for private(i) shared (x,y)
   for (i=0; i<n; i++){
        x[i] = x[i] + y[i];
   }

}
```

```
ENV VAR: OMP_NUM_THREADS

Functions:
   OMP_GET_MAX_THREADS()
   OMP_GET_NUM_THREADS()
   OMP_SET_NUM_TRHEADS()

Scheduling: OMP_SCHEDULE
   STATIC, DYNAMIC,
   GUIDED, RUNTIME
```

Hybrid Architectures: "Clusters" of SMPs

Shared Memory nodes



Hybrid Architectures: Examples

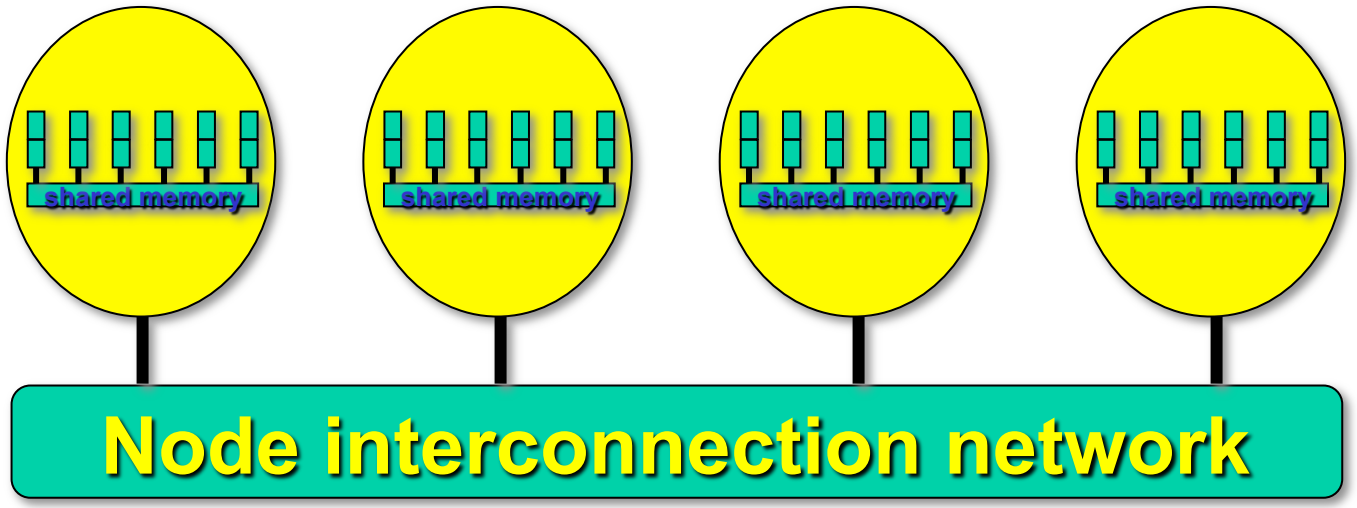
IBM Blue Gene series

- ✓ 1024 SMP nodes per rack
- ✓ 4 cores per SMP node, 2-4 Gbytes per node
- ✓ Hundreds of racks to reach 3PFlops

IBM P775 and new Power8 based systems

- ✓ Multisocket nodes
- ✓ Each node has shared memory

Programming models on hybrid architectures

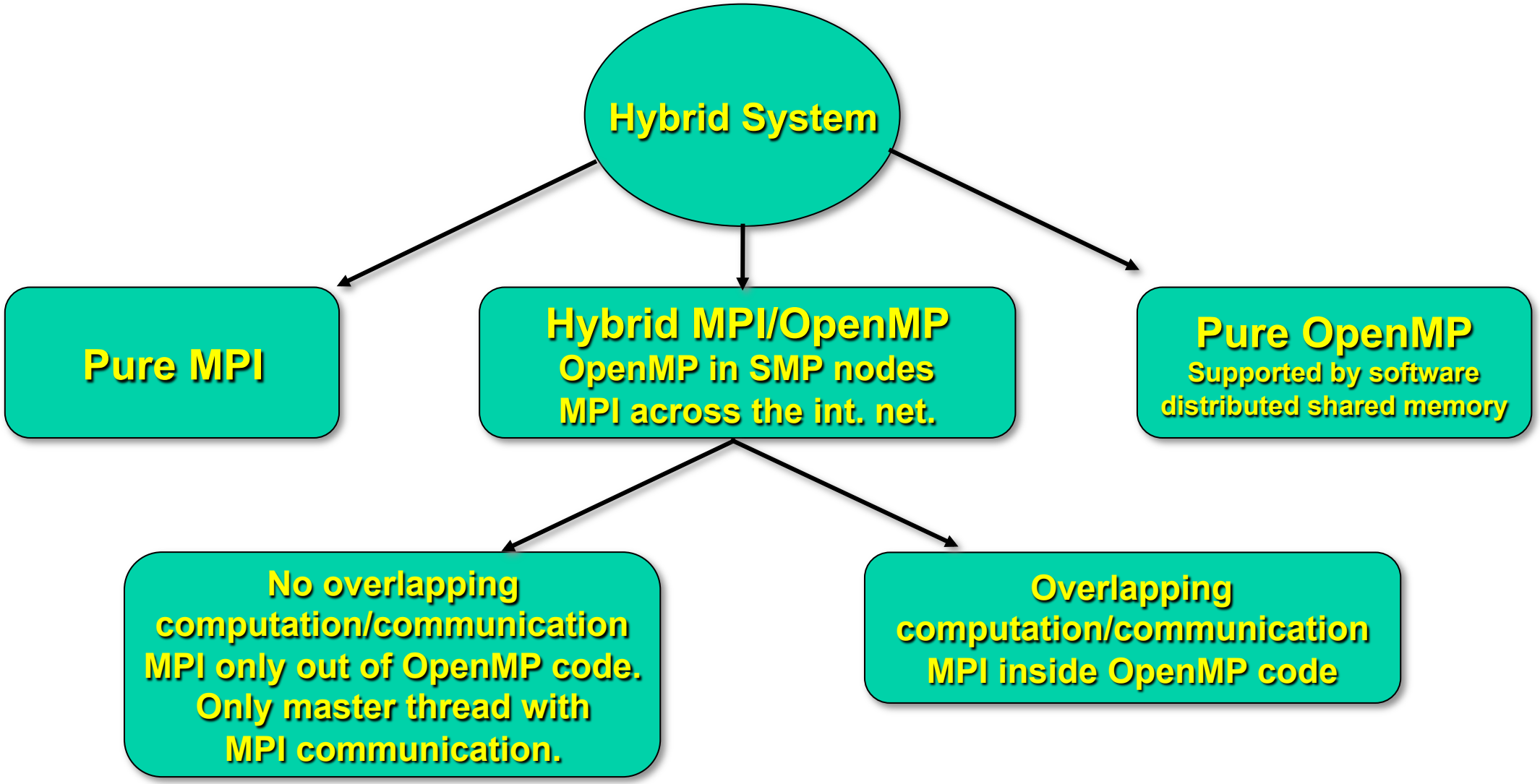


Pure MPI: Remember SMP supports MPI as well. Only MPI processes across the machine

Hybrid MPI/OpenMP: OpenMP inside SMP nodes and MPI across the node interconnection network

Others: Distributed shared memory (runtime hides the inter. net.), HPF,…

Hybrid systems programming hierarchy



Hybrid MPI/OpenMP Example: adding two vectors

```
#include “mpi.h” /* Include MPI header file */
int main(int argc, char **argv){
   int rnk, sz, n, I, info;
   double *x, *y, *buff;
   n = atoi(argv[1]); /* Get input size */

   /* Initialize threaded MPI environment */
   MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &info);
   MPI_Comm_size(MPI_COMM_WORLD, &sz); /* Find out how many MPI procs */

   chunk = n / sz; /* Assume sz divides n exactly */

  MPI_Scatter(buff,chunk,MPI_DOUBLE,x,chunk,MPI_DOUBLE,0,MPI_COMM_WORLD);
   MPI_Scatter(&buff[n],chunk,MPI_DOUBLE,y,chunk,MPI_DOUBLE,0,MPI_COMM_WORLD);

   #pragma omp parallel for private(i,chunk) shared(x, y)
   for (i=0; i<chunk; i++) x[i] = x[i] + y[i];

   MPI_Gather(x,chunk,MPI_DOUBLE,buff,chunk,MPI_DOUBLE,0,MPI_COMM_WORLD);
   MPI_Finalize(); }
```

Hybrid MPI/OpenMP: Use multithreaded libraries

```
#include "mpi.h" /* Include MPI header file */
int main(int argc, char **argv){
   int rnk, sz, n, I, info;
   double *x, *y, *buff;
   n = atoi(argv[1]); /* Get input size */

   /* Initialize threaded MPI environment */
   MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &info);
   MPI_Comm_size(MPI_COMM_WORLD, &sz); /* Find out how many MPI procs */

   chunk = n / sz; /* Assume sz divides n exactly */

   MPI_Scatter(buff,chunk,MPI_DOUBLE,x,chunk,MPI_DOUBLE,0,MPI_COMM_WORLD);
   MPI_Scatter(&buff[n],chunk,MPI_DOUBLE,y,chunk,MPI_DOUBLE,0,MPI_COMM_WORLD);

   DAXPY(&CHUNK, &DONE, X, &INC, Y, &INC);

   MPI_Gather(x,chunk,MPI_DOUBLE,buff,chunk,MPI_DOUBLE,0,MPI_COMM_WORLD);
   MPI_Finalize(); }
```

Hybrid MPI/OpenMPI Initialization modes

```
MPI_Init_thread(&argc, &argv, int mode, &info);
```

mode: Can have four (4) different values

- ✓ **MPI_THREAD_SINGLE (0)** No threads are allowed.
- ✓ **MPI_THREAD_FUNNELED (1)** Threads are allowed. However only the master thread is allowed to call MPI communication primitives
- ✓ **MPI_THREAD_SERIAL (2)** Threads are allowed. All threads can call MPI communication primitives. However communications are scheduled in a serial manner
- ✓ **MPI_THREAD_MULTIPLE (3)** Threads are allowed. All threads can call MPI communication primitives at arbitrary order

info: returns the supported mode of the hybrid system.

Hybrid MPI/OpenMPI Master Only

The Hybrid system has to support at least:

- ✓ **`MPI_THREAD_FUNNELED (1)` Threads are allowed. However only the master thread is allowed to call MPI communication primitives**

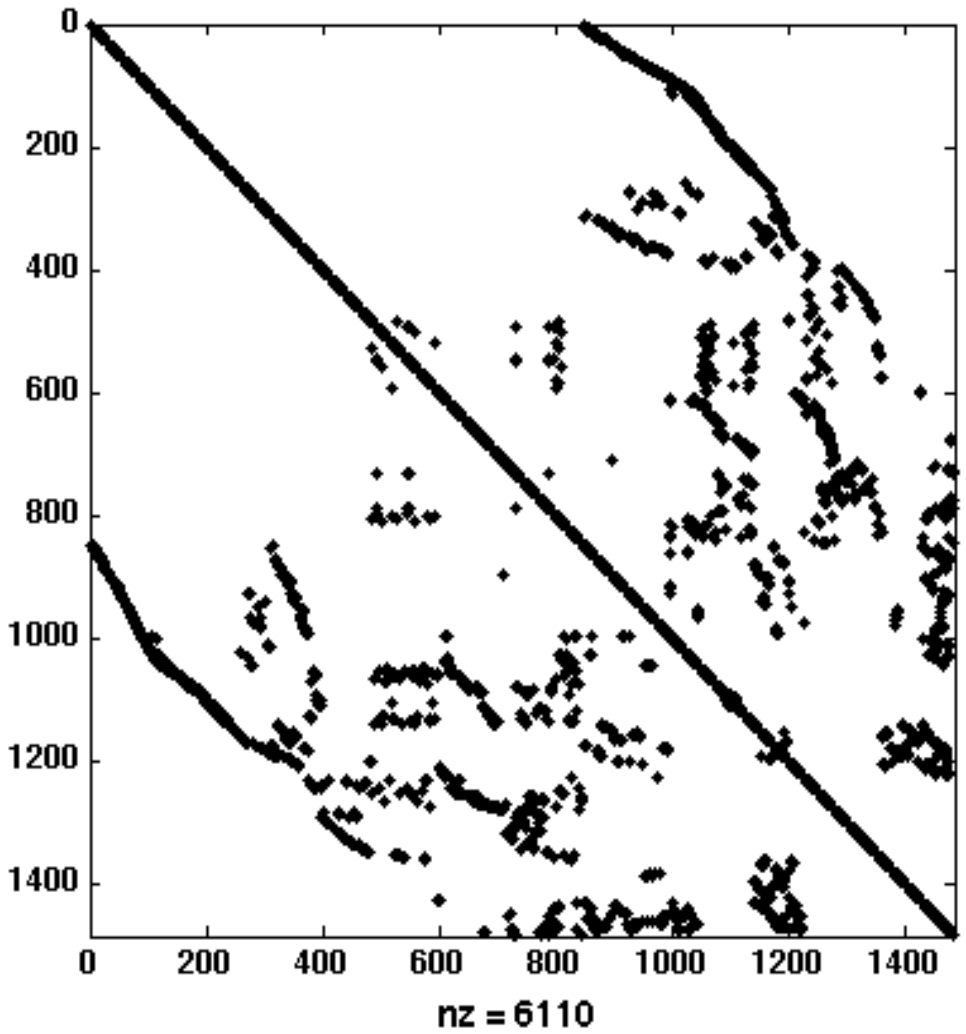
Advantages:

- ✓ **Simple! We can avoid MPI calls within SMP nodes**
- ✓ **Easier mapping of application on the parallel machine**
  - o **Coarse decomposition of load and data is done across the SMP nodes**
  - o **Finer grain computations are easier handled using OpenMP**
  - o **Using the scheduling schemes of OpenMP we can easier deal with load balance issues, avoiding complicated, error prone, MPI software development to handle imbalances**

Problems?

- **Are we under-utilizing the inter-node communication bandwidth?**
- **Are we wasting resources since only master thread communicates while others are sleeping?**

Hybrid MPI/OpenMPI: Load balance example



Sparse Matrix-Vector product

Major kernel in Scient. Comp.

Consider a simple row-wise distribution of the matrix

P=2: min_nz=2544 max_nz=3566

P=4: min_nz=1136 max_nz=1848

P=8: min_nz=464 max_nz=1035

P=16: min_nz=226 max_nz=592

P=32: min_nz=94 max_nz=312

Thus, other sophisticated partitioning schemes are used In MPI applications (Graph. Part: ParMETIS, Zoltan, PaToH, Scotch)

Can MPI/OpenMP help?

Hybrid MPI/OpenMPI: Load balance example

Sparse Matrix-Vector product: Notice that load imbalance situation deteriorates in case of fine grain.

Solution: Keep the coarse grain characteristics by MPI at the inter SMP-node level and perform fine grain parallel computations with OpenMP within the SMP nodes

```
  1) MPI_Scatter sparse matrix A;
  2) MPI_Bcast vector x stored in buff;
  .
  .
  .
  chunk = n / sz; /* Notice sz will be small (# of SMP nodes)  */

  #pragma omp parallel for private(i,chunk) shared(A, x)
  for (i=0; i<chunk; i++)
      # Perform sparse dot product of row A(i,:) and vector x: y = A(:,i)x

  MPI_Gather(y,chunk,MPI_DOUBLE,buff,chunk,MPI_DOUBLE,0,MPI_COMM_WORLD);

  MPI_Finalize();
}
```

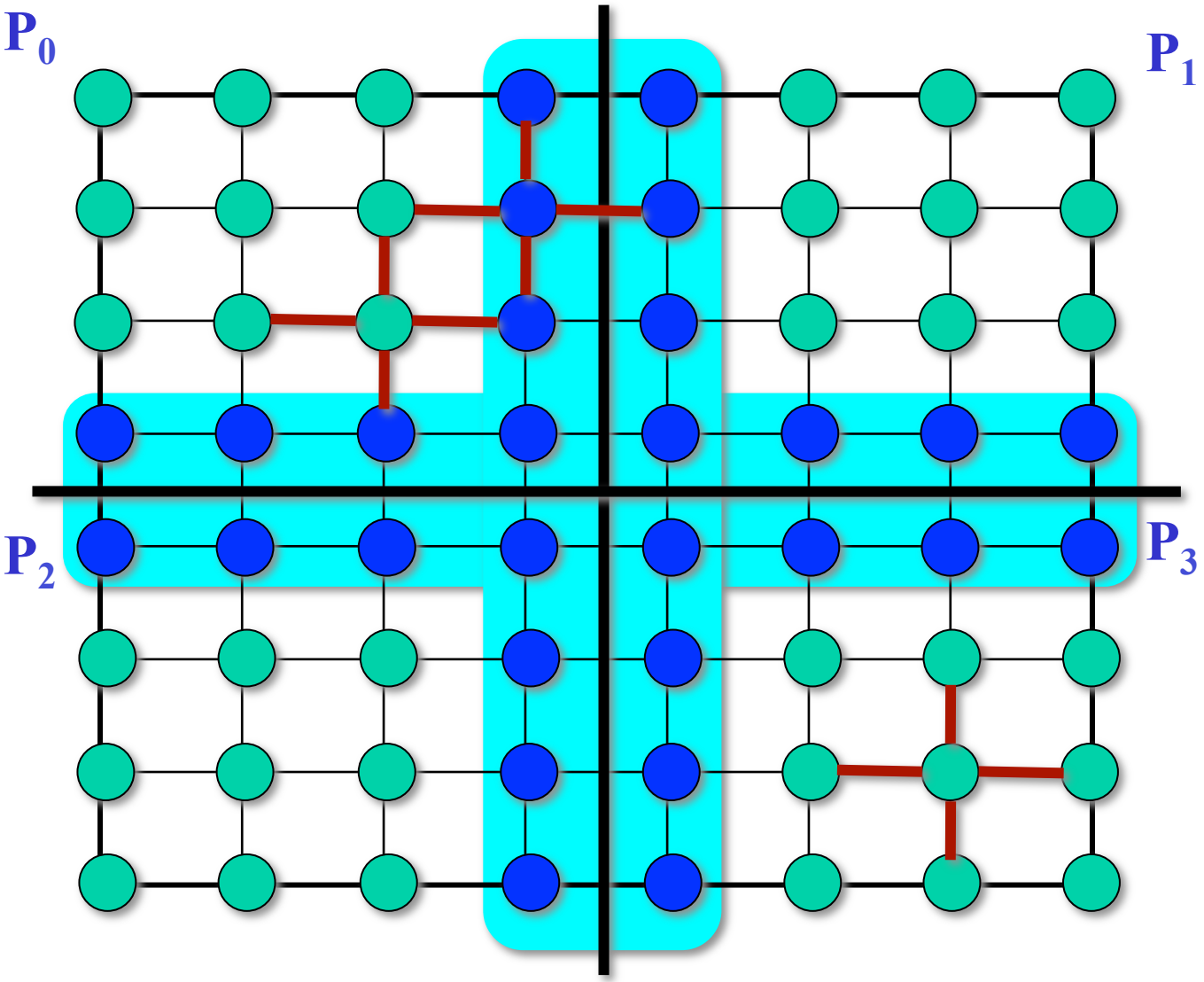
MPI/OpenMPI: Overlapping computation/communication

Not only the master but other threads communicate. Call MPI primitives in OpenMP code regions.

Remember: we need `MPI_THREAD_SERIAL` or `MPI_THREAD_MULTIPLE`

```
if (my_thread_id < # ){
    MPI_… (communicate needed data)
} else
    /* Perform computations that do not need communication */
    .
    .
    .
}
/* All threads execute code that requires communication */
    .
    .
```

Overlapping computation/communication: Example



Suppose we wish to solve the PDE

$$-\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right)=f(x,y)$$

Using the Jacobi method: the value of u at each discretization point is given by a certain average among its neighbors, until convergence.

Distributing the mesh to SMP clusters by Domain Decomposition, it is clear that the GREEN nodes can proceed without any comm., while the Blue nodes have to communicate first and calculate later.

Overlapping computation/communication: Example

```
for (k=0; k < MAXITER; k++){
    /* Start parallel region here */
    #pragma omp parallel private(){
            my_id = omp_get_thread_num();

            if (my_id is given “hallo points”)
                    MPI_SendRecv(“From neighboring MPI process”);
            else{
                    for (i=0; i < # allocated points; i++)
                            newval[i] = avg(oldval[i]);
            }

            if (there are still points I need to do)
                    for (i=0; i< # remaining points; i++)
                            newval[i] = avg(oldval[i]);

            }
            for (i=0; i<(all_my_points); i++)
                            oldval[i] = newval[i];
    }
    MPI_Barrier(); /* Synchronize all MPI processes here */
}
```

Pure MPI v.s. Hybrid MPI/OpenMP and overlapping communications

Fast networks:

- ✓ If we have good mapping of MPI processes on machine topology/architecture
- ✓ If we have achieved good load balance…then
- ✓ Pure MPI is almost always faster…

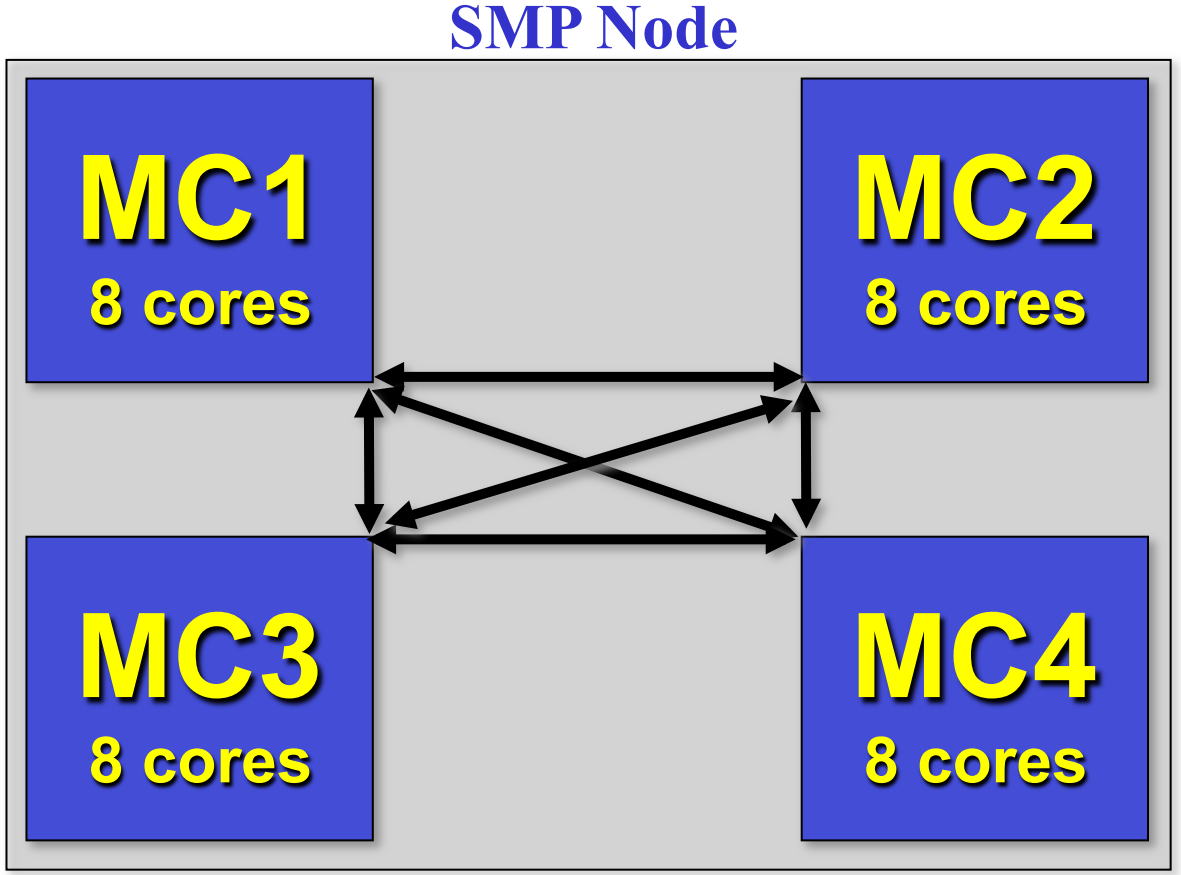
Slow (or even ad-hoc) networks

- ✓ MPI mapping less important
- ✓ Careful selection of communicating threads important
- ✓ Good when long MPI messages are expensive

Intermediate situations also exist:

- ✓ Several MPI processes on the same SMP node
- ✓ Easier to saturate interconnection node (keep close to max performance)
- ✓ Some of the mapping problems come back though! (See next slide)

Hybrid MPI/OpenMP: MPI Process/threads binding



MPI process placement

- ✓ MPI processes should be placed so that threads that belong to a single MPI process share neighboring memory cells
- ✓ MC: Multi-chip module
  MCs hold 4 Chips:
  For 4 MPI processes put each one of them on a separate MC
- ✓ Do not put all MPI processes on one MC (say MC1)

Topology and Application mapping problems overview

Pure MPI

- ✓ Topology issues. How to best map application on complex hybrid topology
- ✓ Need shared memory implementation for intra-node MPI intrinsics?

Hybrid MPI/OpenMP

- ✓ Too many threads in the node can saturate the intra-node network
- ✓ If no overlapping computation/communication: wasted resources by sleeping threads
- ✓ Need for careful data placement needed for OpenMP?
- ✓ Thread creation? Thread synchronization? All this is overhead!

Overlapping communication/computation

- ✓ Code complexity: careful mapping, load balancing?
- ✓ How to select how many threads to allocate to "hallo" data
- ✓ Good knowledge of applications/hardware is needed

Conclusions

Hybrid MPI/OpenMP

- ✓ Natural paradigm for clusters of SMP's
- ✓ May offer considerable advantages when application mapping and load balancing is tough
- ✓ Benefits with slower interconnection networks (overlapping computation/ communication)

- Requires work and code analysis to change pure MPI codes
- Start with auto parallelization?
- Link shared memory libraries…check various thread/MPI processes combinations
- Study carefully the underlying architecture

- ✓ What is the future of this model? Could it be consolidated in new languages?
- ✓ Connection with many-core?

…stay tuned

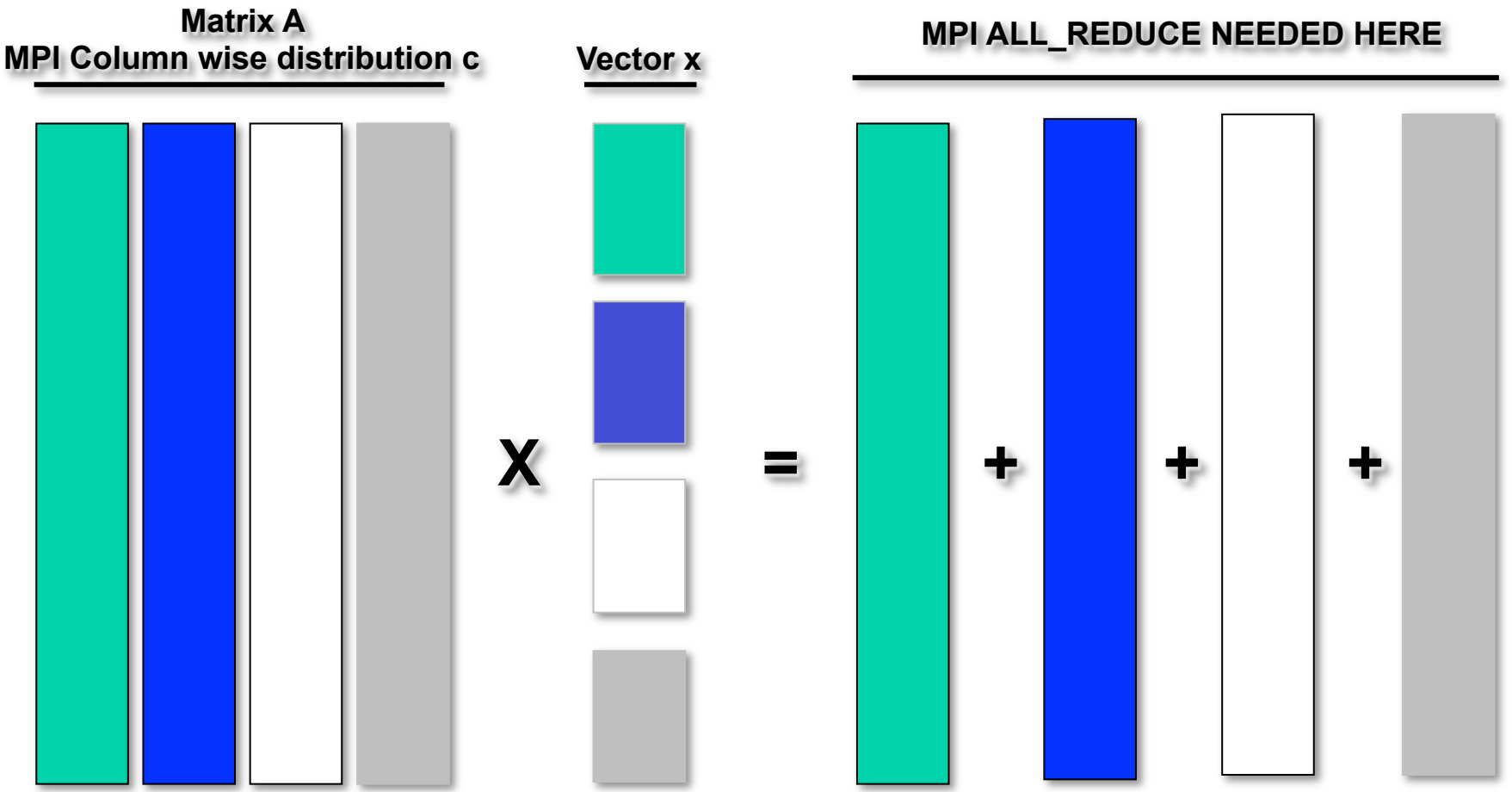
Example 1 : Simple Hello world!

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv){

     /* Initialize threaded MPI */
     MPI_Init_thread(&argc, &argv, THREAD_MODE(?), &info);

     /* Find and print out thread mode (from info) */
     MPI_Comm_rank(…
     MPI_Comm_size(…
     printf("My rank is %d …

     /* Get number of requested threads from argv[1] */
     /* Set number of threads omp_set_num_threads();… */

     /* Start a parallel region: use #pragma omp parallel */
     {
               Each thread print its id (so get it! …omp_get_num_thread()) and
     "Hello world"
     }
     /* Finalize mpi */
}
```

Example 2: Simple global summation!

```
#include <stdio.h>
#include <stdlib.h>
#include “mpi.h”
int main(int argc, char **argv){

     /* Initialize threaded MPI and print out MPI-OpenMP
        information as before */
     MPI_Init_thread(&argc, &argv, THREAD_MODE(?), &info);
     .
     .

     /* Set number of desired threads */

     /* Start a parallel region: use #pragma omp parallel */
     {
     /* Each thread does MPI_ALLREDUCE(“MPI_SUM”) on its thread id */
     Prototype: int MPI_Allreduce ( void *sendbuf, void *recvbuf, int count,
     MPI_Datatype datatype, MPI_Op op, MPI_Comm comm )

     /* Each thread printout its MPI rank, its thread is and the global sum, in
         a single line */ }
     /* Finalize mpi */
}
```

Example 3: Matrix-Vector multiplication (column-wise version)

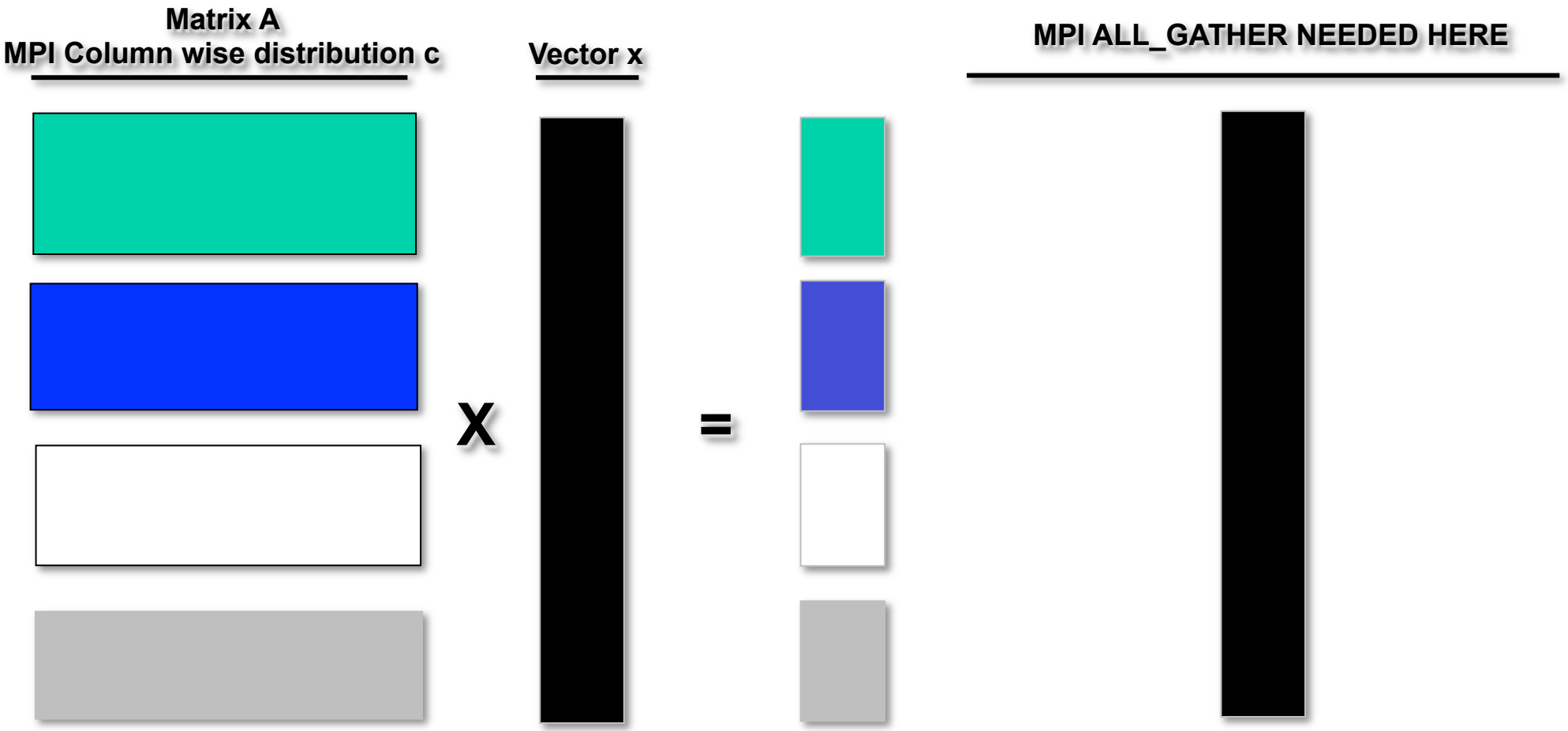


Each local matrix-vector is to be done with 2 nested loops. Use OpenMP to parallelize these loops

Example 3: Matrix-Vector multiplication (row-wise version)

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
int main(int argc, char **argv){

      /* Initialize threaded MPI and print out MPI-OpenMP information as before */
      MPI_Init_thread(&argc, &argv, THREAD_MODE(?), &info);
      /* Get size of matrix. Assume square */
      /* Set number of desired threads */
      /* Find out what is the chunk size (#of columns) each MPI proc gets */
      /* Make sure load imbalance is only +1! How do you do this? */
      /* Initialize local matrices to 1s to check accuracy of results*/

      /* for loop for the local matrix-vector product */
      # omp parallel for private(?) shared (?) /* (here or in the inner loop?) */
      for (i=0; i<n; i++){
         for (j=0; j<chunk; j++){
                  . . .
         {
      }
      Do global summation to get the result back to all MPI procs!
      /* Finalize mpi */
      Try to get timing information. Use MPI_Wtime. What is the min and max time across
      MPI procs?
}
```

Example 4: Matrix-Vector multiplication (row-wise version)

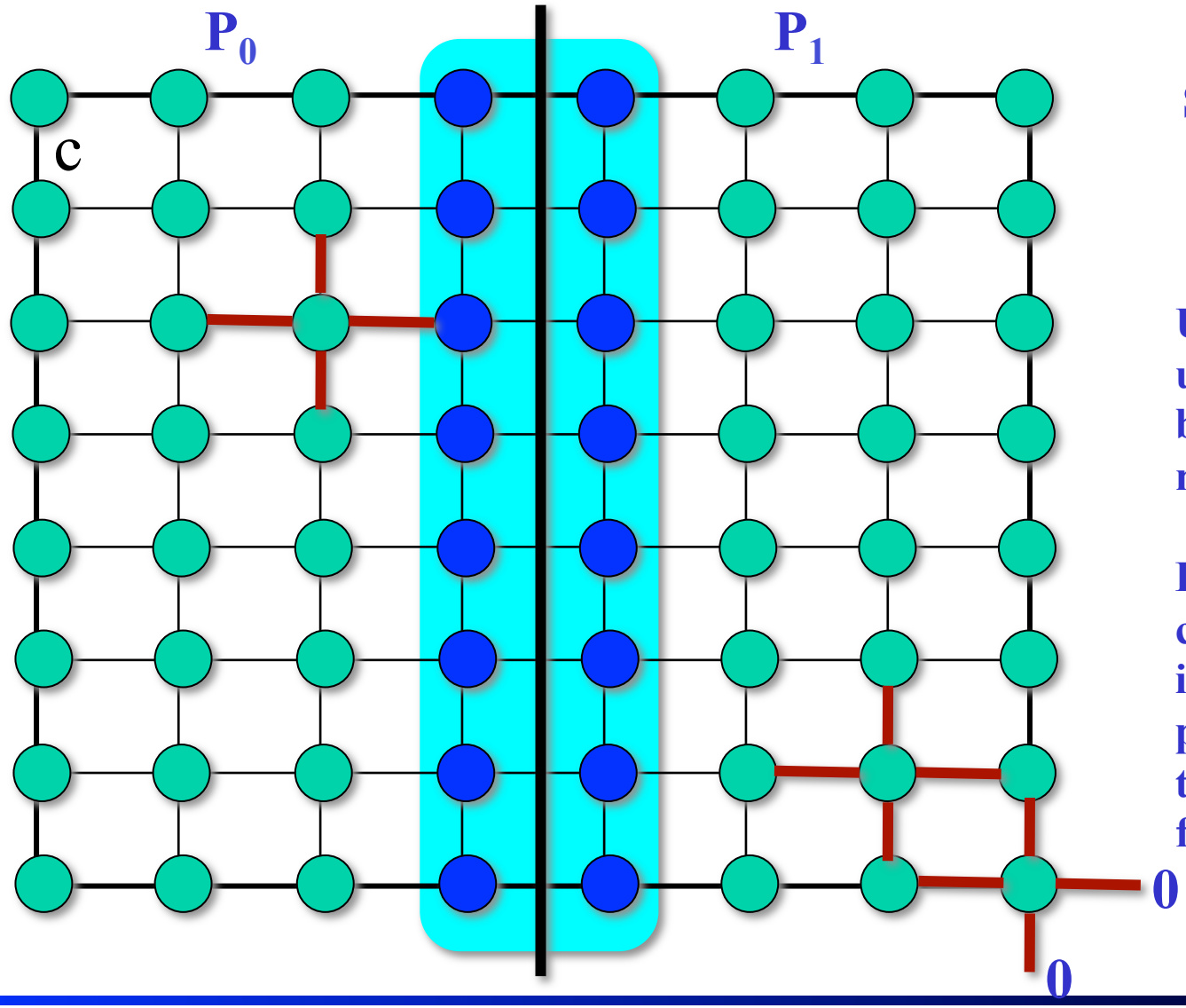


Each local matrix-vector is to be done with 2 nested loops. Use OpenMP to parallelize these loops

Example 4: Matrix-Vector multiplication (row-wise version)

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
int main(int argc, char **argv){

      /* Initialize threaded MPI and print out MPI-OpenMP information as before */
      MPI_Init_thread(&argc, &argv, THREAD_MODE(?), &info);
      /* Get size of matrix. Assume square */
      /* Set number of desired threads */
      /* Find out what is the chunk size (#of columns) each MPI proc gets */
      /* Make sure load imbalance is only +1! How do you do this? */
      /* Initialize local matrices to 1s to check accuracy of results*/

      /* for loop for the local matrix-vector product */
      # omp parallel for private(?) shared (?) /* (here or in the inner loop?) */
      for (i=0; i<chunk; i++){
         for (j=0; j<n; j++){
                  . . .
         {
      }
      Do global GATHER to get the result back to all MPI procs!
      /* Finalize mpi */
      Try to get timing information. Use MPI_Wtime. What is the min and max time across
      MPI procs?
}
```

Example 5. Overlapping computation/communication



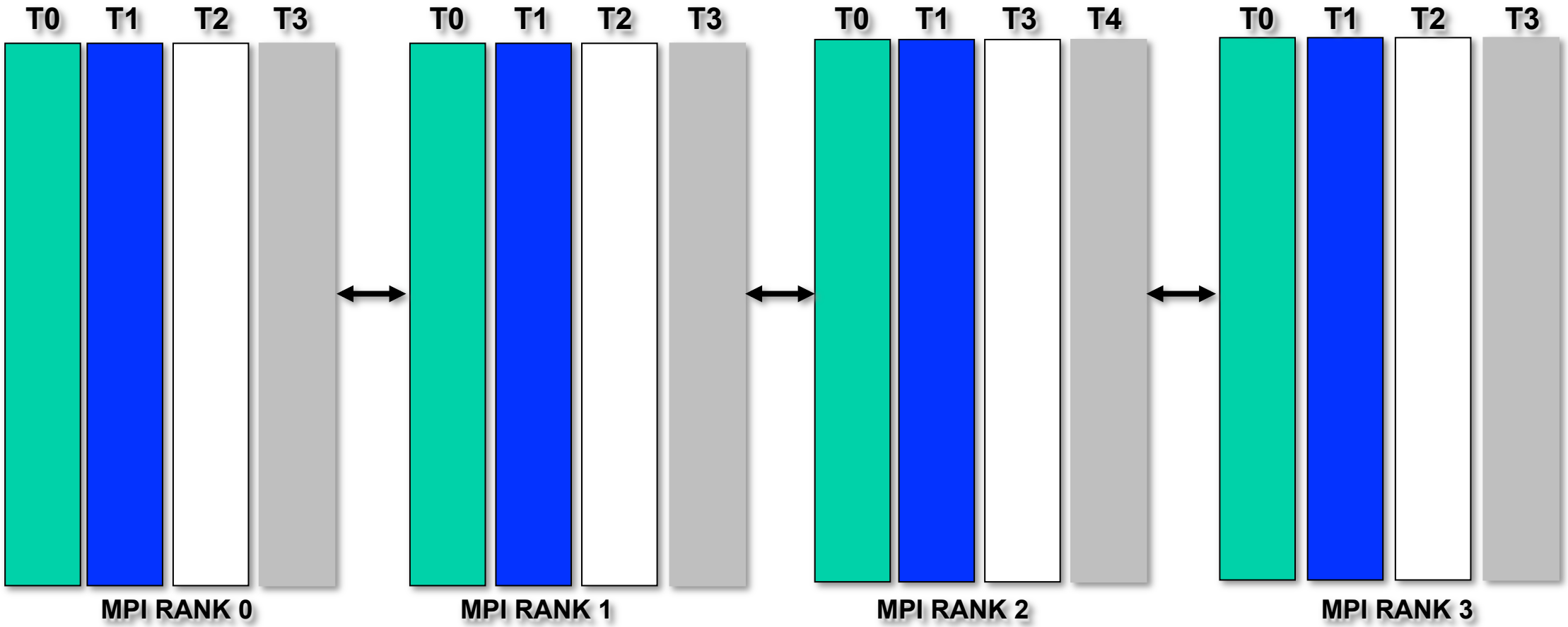
Suppose we wish to solve the PDE

$$-\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right)=f(x,y)$$

Using the Jacobi method: the value of u at each discretization point is given by a certain average among its neighbors, until convergence.

Distributing the mesh to SMP clusters by Domain Decomposition, it is clear that the GREEN nodes can proceed without any comm., while the Blue nodes have to communicate first and calculate later.

Example 5: Overlapping computation-communication



Each MPI proc holds a number of consecutive columns of matrix A. For each entry of A we need to substitute it with the mean of its 4 neighbors and itself. Out of matrix elements are considered to be equal to zero.

Observe: At each MPI proc, threads T0 and T3 need "hallo" data before they can proceed. The other threads can proceed directly! You will use arbitrary number of MPI procs and arbitrary number of threads. For testing: Initialize each MPI proc chunk with the value (RANK+1). Check after the initial communication test that the values have indeed been exchanged!

Example 5: Overlapping Computation-Communication

```
/* Initialize threaded MPI and print out MPI-OpenMP information as before */
MPI_Init_thread(&argc, &argv, THREAD_MODE(?), &info);
/* Get size of matrix. Assume square */
/* Set number of desired threads. Get form input */
/* Find out what is the chunk size (#of columns) each MPI proc gets */
/* Make sure load imbalance is only +1! How do you do this? */
/* Initialize local matrices to RANK+1 to check accuracy of results*/
/* Assume an external loop */
for (k=1; k<max_iters; k++){/* Get max_iters from the input */
    /* Start parallel region here */
    #pragma omp parallel private(?) /* What else? */
    {
              /* Find out what is your thread id */
              myid = omp_get_thread_num();
              /* Find out which columns of the local chunk of matrix A, I am handling
                 Assume static, chunk like distribution */
              #pragma omp barrier /* Synchronize threads here! */

              /* if I need "hallo" data then do communication. First send then receive */
              if (myid=(thr-1) && (RANK<(SZ-1)){/* I am last thread but not last MPI proc */
                        MPI_send: my last column to my right MPI neighbor
              }
              if (myid=0 && (RANK>0)){/* I am first thread but not first MPI proc */
                        MPI_recv: to buffer column from my left MPI neighbor
              }
              /* Now reverse roles! Sender becomes receiver and vice-versa!!!! */
              . . .
              /* Do all the useful calculations here! For this test just print the exchanged values */
    }
}
MPI_Finalize(); }
```